Design Pattern: It is at code level, like factory, Singleton

Architecture Pattern: Is overall layout, a high-level diagram, solution architecture like MVC, MVVM,

Architecture Principles/Style: Principles follow in the solution like Rest, SOA, IOC

Design patterns are best practices for object-oriented programming or solution to common software design problems.

Entities are objects or nouns. So identify entities and create classes and objects. Identify Pronouns and create properties of the object. And actions or verbs becomes method. There are two kinds of relationship between entities, ‘is a’ and ‘uses’. Shiv is a son of his father, it is a parent-child relationship. Shiv uses car or has a car, ‘uses’ relationship.

So for ‘is a ’ relationship you can use base class to point both type

|  |  |
| --- | --- |
| OOP Phase | Pattern |
| Template/Class creation | Structural |
| Class Instantiation | Creational |
| Runtime | Behavioural |

To test good software architecture look at how architecture handles changes.

Composition is flexibility and inheritance is structure, you have to balance both to get good software. If you go full composition, you will end up having set of randon things flying all over. And if you go full inheritance, you will end up having rigidity, you can’t change.

RIP pattern : Replace If with polymorphism, Instead of using if and creating instance based on condition. Use a dictionary to get instance of the base class type.

Lazy loading : A concept where we delay the loading of the object until the point where we need it. Load objects on demand rather than loading it unnecessarily. Like if a dictionary is needed then only add items in it. Don’t add it at declaration or in constructor.

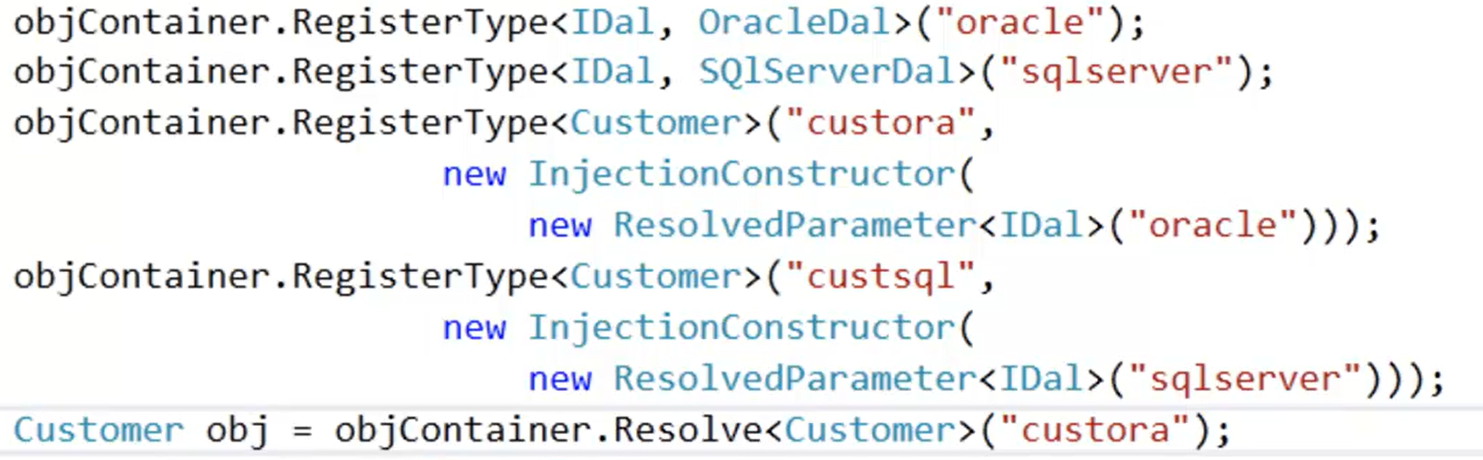
It minimizes the start up time of the application. Application consumes less memory. Unnecessary execution can be avoided.

It becomes complicated sometimes, as we need to check if the loading is needed or not, which causes a slight decrease in performance.

Eager loading : Opp. Of lazy loading is eager loading.

**IOC (Inversion Of Control) :** It’s a principle where we move unwanted work or work which affects a class to some other entity, which takes that responsibility in more better way, is termed as inversion of control. Class takes care of concerned logic only and unconcerned logic is delegated or inverted to some other entity. It benefits as decouple system and cleaner code. It can be implemented using dependency injection, events, delegates, service locator.

**Dependency Injection:** It is a technique which helps to inject dependent objects of a class, thus making architecture loosely coupled. A container is used to register types and to get resolved objects. Microsoft.Practices.Unity can be used. On registering types, in case of multiple registration of same interface type, it will return last registered class mapped to that interface. Or you can map types in the xml and load configuration from there.



**Single Responsibility Principle :** A class should have single responsibility. It should not try to do lot of things.

**Separation Of Concern :** Unnecessary concern which are with the entity should be separated to some other thing

**Creational Pattern** It is all about different ways to create objects, helps to centralize object creation process and used by client to get objects. It prevents client to use concrete classes.

**Structural Pattern :** Relationship between objects. Suppose customer class is using phone class. So there is a relationship b/w customer and phone class. This pattern helps you solve the structural issues.

**Behavioural Pattern :** When you want to change the behaviour of an object and you don’t want this change affect the project very much.

All factory patterns help you control and centralize the object creational process. If you are creating same instance at many places, then for a single change, you have to modify it at all the places. So, by centralizing the object creational code, you just have to change it at one place.

**Simple Factory Pattern :** Centralize new keyword. It helps to centralize object creation and thus helps to achieve a decoupled system. Simply creating instance of a particular interface or base class without exposing the instantiation logic.

**Abstract Factory Pattern**, provide an interface for creating families of related or dependent objects. Here we add one more level to factory, and factory is created using a factory or we are abstracting a factory using a factory.

Abstract factory, concrete factory, abstract product, product

**Factory method pattern**, defines an interface for creating objects, but let subclasses to decide how to instantiate.

Product, concrete product, creator, concrete creator

**Singleton Pattern** When we want only one instance of the object to be created and shared b/w the clients. And no client can create an instance from outside. Steps to make as class as static :

1. Define the constructor as private
2. Defined the instance and methods as static

**Builder Pattern,** separates the construction of a complex object from its representation so that the same construction process can be used for different representation.

Builder(abstract interface), concrete builder(actual construction code), Director(construct object using builder), Product(complex object)

Singleton : also known as anti pattern

**Behavioral Pattern**

**Strategy Pattern :**  This pattern defines family of algorithms and anyone can be easily interchangeable with each other. It helps to choose algorithms dynamically. For ex. You can inject different validation algorithm to validate a class. It also helps you apply IOC, or SRP or SOC. For ex, you define discounts for each day of week and inject it based on the current day of the week.

**Template Pattern :** A template is used to defined the structure or sequence and child class is used to define how the sequence will behave. For ex. In an abstract class a method is defined which execute set of methods in a sequence. And one of the method is marked abstract, which is defined by child classes. In that way, a template is created and child classes are using that template and defining how the method will behave. We have a parent class where we have a fixed sequence, but child classes can override individual behaviour of the method but cannot changed the sequence. Like we have a page cycle of asp.net. Where we can override methods and implement our own logic

**Repository Pattern**  Whole goal of pattern is for decoupling dal from models

- first create a inteface, and all client will consume this interface

- then create a common abstract class for dal

- create a dal class with all the data logic